# Do you want to play a game?

Fredrik Carlsson
Katariina Martikainen
Giorgos Tagkoulis

KTH Kista 13.11.2018

# Kernel Modules

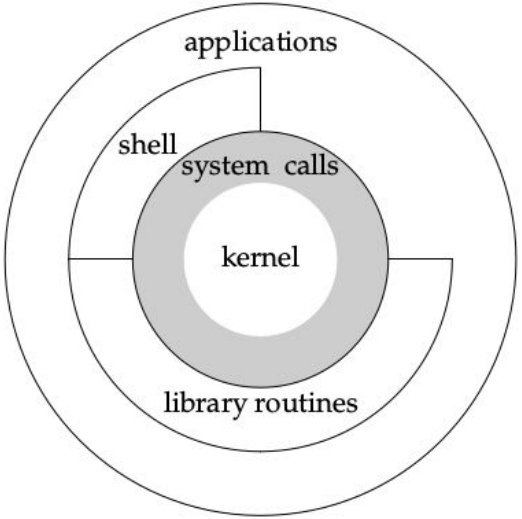
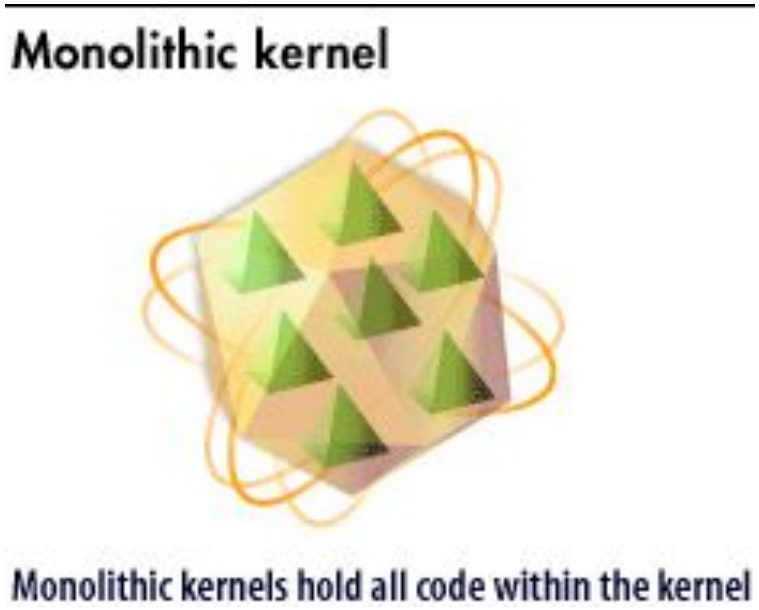



**Figure 1.1** Architecture of the UNIX operating system



# Character & Block Devices

- Device driver = a piece of software that operates or controls a particular type of device
- Device file = an interface for a device driver that appears in a file system as if it were an ordinary file

- A **Character** ('c') **Device** is one with which the Driver communicates by sending and receiving single **characters** (bytes, octets).
- A **Block** ('b') **Device** is one with which the Driver communicates by sending entire **blocks** of data.

# Headerfiles

- Used for creating a public interface during **#include**
- Best be named **identical** to its corresponding **C/C++** file
- Fil extension ***.h***

```
1  #define CONSTANT 42
2
3  void fooBar(int x);
```

*Example of a headerfile*
*TestFile.h*

*When included it defines the function **fooBar(int x)** and **CONSTANT.***
*So they can be called upon from the file doing the import.*

# Headerfiles - Example

```c
#include "TestFile.h"

int main(){
  fooBar(CONSTANT);
}
```

***Main.c***

```c
#include <stdio.h>

void fooBar(int x){
  printf("Number %i \n", x);
}
```

***TestFile.c***

```c
#define CONSTANT 42

void fooBar(int x);
```

***TestFile.h***

```
Main.c  TestFile.c  TestFile.h
fredrik@fredrik-VirtualBox:~/Documents/Test$ gcc -o Test Main.c TestFile.c
fredrik@fredrik-VirtualBox:~/Documents/Test$ ./Test
Number 42
```

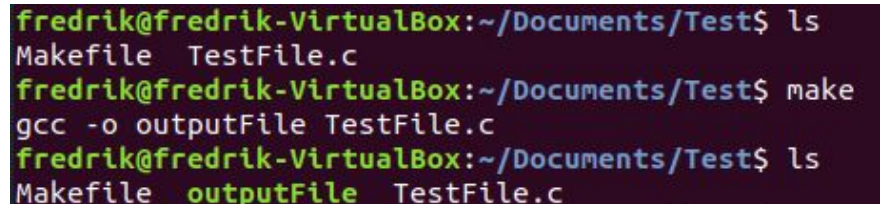
*Compilation of **Main.c** and **TestFile.c***

# Makefile

- File containing instructions of how to compile and link
- File must be named "***Makefile***"
- Activated using the "***make***" command
- ***make*** can be installed using "***sudo apt install make***"
- Speciell syntax, [more info here]

```
1  all:
2      gcc -o outputFile TestFile.c
```

*Example **Makefile***




*When **make** is called it compiles **TestFile.c** into **outputFile***

# Return Values with Buffers

```
int n1, n2;
if(getMyMsg, &n1, &n2) != -1){
  print("I got some numbers %i %i", n1, n2);
}
else{
  printf("Something went wrong\n", );
}
```

```
int getMyMsg(int* output2, int* output3){
  if(true){
    *output1 = 10;
    *output2 = 15;
    return 0;
  }
  return -1;
}
```

# Nifty Syntax

- **printk** *// Prints to the kernel log*
- **ioctl** *// Sends a request to a kernel module*
- **less** *// Terminal text viewer*
  - **‘shift’** + ‘**g**’ *// Goes to the end of text*
  - ‘**:**’ + ‘**q**’ *//Exits less*

# 2017-08-21

We can create so called *character device* and interact with it using `ioctl`. In the code below, describe what `fd`, `JOSHUA_GET_QUOTE` and `buffer` is and how the *device* could work.

```
if (ioctl(fd, JOSHUA_GET_QUOTE, &buffer) == -1) {
  perror("Hmm, not so good");
} else {
  printf("Quote - %s\n", buffer);
}
```

# 2017-08-21

We can create so called *character device* and interact with it using `ioctl`. In the code below, describe what `fd`, `JOSHUA_GET_QUOTE` and `buffer` is and how the *device* could work.

```
if (ioctl(fd, JOSHUA_GET_QUOTE, &buffer) == -1) {
  perror("Hmm, not so good");
} else {
  printf("Quote - %s\n", buffer);
}
```

**Answer**:

**fd** is a file descriptor that we get when we open the file that the module has been registered to.

**JOSHUA_GET_QOUTE** is an encoded instruction that describes what we wished to do, if we have inputted a buffer, and if it's to be used for reading or writing.

**buffer** is a memory area where the module can read or write to.

In this example it's reasonable to think that we request a quote and that the module will write this to **buffer**

# 2017-01-14

If we should add a functionality to a Linux kernel, we must recompile the kernel and restart the system - true or false? Motivate your answer.

# 2017-01-14

If we should add a functionality to a Linux kernel, we must recompile the kernel and restart the system - true or false? Motivate your answer.

**Answer:** False - you can load a kernel module into a running kernel using the command `insmod`.

# 2017-01-14

There are several ways in which we can communicate with at kernel module, the code below shows one way. Here we initialize certain data structures that then allow other processes to easily access the module. Which is the visible interface that processes will use? Give a brief explanation.

```
static int __init skynet_init(void) {
  proc_create("skynet", 0, NULL, &skynet_fops);
  printk(KERN_INFO "Skynet in control\n");
  return 0;
}

static void __exit skynet_cleanup(void) {
  remove_proc_entry("skynet", NULL);
  printk(KERN_INFO "I'll be back!\n");
}
```

# 2017-01-14

There are several ways in which we can communicate with at kernel module, the code below shows one way. Here we initialize certain data structures that then allow other processes to easily access the module. Which is the visible interface that processes will use? Give a brief explanation.

```
static int __init skynet_init(void) {
  proc_create("skynet", 0, NULL, &skynet_fops);
  printk(KERN_INFO "Skynet in control\n");
  return 0;
}

static void __exit skynet_cleanup(void) {
  remove_proc_entry("skynet", NULL);
  printk(KERN_INFO "I'll be back!\n");
}
```

**Answer:** We register the module as a file under `/proc`. Processes can now use regular file operations to communicate with the module.

# Things to ramble about

- Makefiles
- Headerfiles
- Kernel Modules
- Character & Block Devices
- System Calls (They should know this now?)
- Pipes?

- printk
- Less:
  - Goto end: “***G***”
  - How to quit: “**:q**”
- ioctl
- file descriptor